Optimization

"Coding Standards? or Manual Optimization?"

Optimization

- Minor changes in how a program is written can make a large difference in how the code is optimized.
- Trade offs between
  - Code readability, maintainability, memory management
- Compiler optimization
  - should not alter program behavior

Example

```
void  foo1(int*  p, int* q){
      *p += *q;
      *p += *q;
}
void  foo2(int*  p, int* q){
      *p += 2*(*q);
}
```

*p = 4 * (*p)

*p = 3 * (*p)

- are foo1 and foo2 equivalent?
  - Memory aliasing

Optimizing Code

- Two ways to optimize
  - **Let the compiler do the optimization**
  - Use: gcc -O file.c -o file

  Eg: for (i=0;i<10000;i++)

  for (j=0;j<i; j++) { i++; i--;}

  runs factor x times faster when compiled with -O flag.
  - **Programmer optimized code**
    - Identify code that can be manually optimized
- We will look some examples of manual optimization

Reduce Variable Scope

```
int bad (void) {
      int i; int j;
      for (i=0; i<500; i++)
          { j = foo();
            if (j > 0)
               return (j);
          }
       return (0);
}
```

```
int good (void) {
       int i;
       for (i=0; i<500; i++)
         { int j;
           j = foo();
           if (j)
              return (j);
          } return (0);
}
```

Compiler can optimize better if the scope of a variable is narrow

Using Natural Sized Variables

```
int  foo ( ) {
      short i= 200, j;
      for (j = 0; j <100; j++)
         if (i == j)
              i = i / j ;
      return I;
}
```

```
int foo2(void) {
      int i = 200;
      for (i=0; i<100; i++)
         if (i== j)
              i /= j;
      return i;
}
```

In foo, the division i = i/j is performed in int mode. So I must be extended from say 16-bit to 32-bits. Also the comparison

(i ==j) requires type conversions.

In foo2, these problems are taken care of. This is an example of trading memory for speed.

Arrange data fields better in a struct

- That is, bigger fields first

Eg: struct {

```
    long int x;
    short int y;
    short int z;
};
```

Why? For optimal alignment for compactness and reduced access times.

Optimizing the code

Compare following codes:

```
void foo( int *data ) {
      int i;
      for(i=0; i<N; i++)
            {  foo2( *data, i);     }
}
```

```
void foo( int *data ) {
   int i;
   int localdata = *data;
      for(i=0; i<N; i++)     {
         foo2( localdata, i);
    }
}
```

Why is this better? If we assume that local data is never changed, it is better to leave it as a local variable. Otherwise it will be copied every time the function foo2 is called (since data is referenced by a pointer)

Optimization ctd…

Use unsigned and/or register integers when you can

Faster arithmetic with some processors. Also good for bit operations

register unsigned int x = 10;

Combine loops:

for (I = 0 ; I< N; I++) task1();

for (I = 0 ; I< N; I++) task2();

If we assume task1 and task2 are independent tasks (task2 doesn't depend on a result from task1)

Better: for (I = 0 ; I< N; I++) { task1(); task2();}

Loop unroll

for (I=0;I<3;I++) x(I);

Better:

x(0); x(1); x(2);

```
for(i=0; i<n; i+=2)
{ C[i]=A[i]+B[i]
  C[i+1]=A[i+1]+B[i+1]
```

Optimization ctd…

Faster Loops

for(i=0; i<N; i++){ … }

Better:

for(i=10; i--;) { … } -- if order does not matter. It is better to say is I non-zero than checking i-N non zero. most target processors will provide decrement-and-branch-if-zero type functionality into their instruction sets

When possible pass a structure by reference

address

Break from a loop early if possible

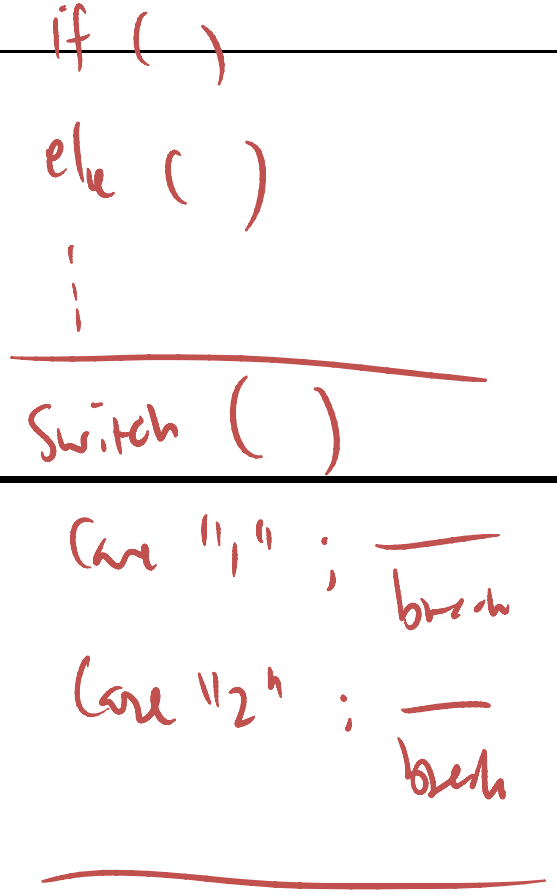
Trade memory for speed

Switch instead of if-else-if

If last case is required in if-then-else then it will test all conditions before getting there. So switch is better

Move unmodified variables out of the loop

```
Int foo(int x, int y){
    int I;
   for (I=0; I<10; I++)
     {  sum = I + x*y; }
}
```



Optimization ctd…

Single dimensional arrays are faster than multidimensional.

*A[I][j] = *(*A + n*I + j) I =1..m, J = 1..n*

Better:

*A[I] = *(*A+I) I = 1…..mn*

Floating point multiplication is faster than division.

*Use: x*0.5 instead of x*1.0/2; or x/2*

Use macros whenever possible. Function calls are expensive

Addition is quicker than multiplication

Avoid using sqrt if possible. It is an expensive function

If (sqrt(x) < 3)

Better:

*If (x*x < 9)*

#define max((a),(b)) ((a)>(b)? a:b)

int max(int x, int y){
if (x>y) return x;
else return y;
}

int z = max(x,y)

int z = (x>y ? x:y));

Overwriting functions

- We can overwrite standard library functions such as sqrt and printf
- But you need to know the exact signature of the standard library function
- An Example

```
#include <stdio.h>
#include <math.h>
int printf(const char* s, ...){ int x = 1;}
double sqrt(double x){ return x*x;}
int main(void){
  int x = 3;
  if ( x == 0 ) printf("%3d\n", x+1);
  else printf("%.3f\n", sqrt(x-1));
  return 0;
}
```